14

Using dependency injection

This chapter covers

- Understanding how dependency injection allows components to access shared services
- Configuring services lifecycles to control when services are instantiated
- Understanding how to define and access services using dependency injection

Services are objects that are shared between middleware components and endpoints. There are no restrictions on the features that services can provide, but they are usually used for tasks that are needed in multiple parts of the application, such as logging or database access.

The ASP.NET Core *dependency injection* feature is used to create and consume services. This topic causes confusion and can be difficult to understand. In this chapter, I describe the problems that dependency injection solves and explain how dependency injection is supported by the ASP.NET Core platform. Table 14.1 puts dependency injection in context.

Table 14.1. Putting dependency injection in context

Question	Answer
What is it?	Dependency injection makes it easy to create loosely coupled components, which typically means that components consume functionality defined by interfaces without having any firsthand knowledge of which implementation classes are being used.
Why is it useful?	Dependency injection makes it easier to change the behavior of an application by changing the components that implement the interfaces that define application features. It also results in components that are easier to isolate for unit testing.
How is it used?	The `Program.cs` file is used to specify which implementation classes are used to deliver the functionality specified by the interfaces used by the application. Services can be explicitly requested through the `IServiceProvider` interface or by declaring constructor or method parameters.
Are there any pitfalls or limitations?	There are some differences in the way that middleware components and endpoints are handled and the way that services with different lifecycles are accessed.
Are there any alternatives?	You don't have to use dependency injection in your code, but it is helpful to know how it works because it is used by the ASP.NET Core platform to provide features to developers.

Table 14.2 provides a guide to the chapter.

Table 14.2. Chapter guide

Problem	Solution	Listing
Obtaining a service in a handler function defined in the `Program.cs` file	Add a parameter to the handler function.	15
Obtaining a service in a middleware component	Define a constructor parameter.	16, 30–32
Obtaining a service in an endpoint	Get an `IServiceProvider` object through the context objects.	17
Instantiating a class that has constructor dependencies	Use the `ActivatorUtilities` class.	18–20
Defining services that are instantiated for every dependency	Define transient services.	21–25
Defining services that are instantiated for every request	Define scoped services.	26–29
Accessing configuration services before the `Build` method is called in the `Program.cs` file	Use the properties defined by the `WebApplicationBuilder` class.	33
Managing service instantiation	Use a service factory.	34, 35
Defining multiple implementations for a service	Define multiple services with the same scope and consume them through the `GetServices` method.	36–38
Using services that support generic type parameters	Use a service with an unbound type.	39

14.1 Preparing for this chapter

In this chapter, I continue to use the `Platform` project from chapter 13. New classes are required to prepare for this chapter. Start by creating the `Platform/Services` folder and add to it a class file named `IResponseFormatter.cs`, with the code shown in listing 14.1.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from https://github.com/manningbooks/pro-asp.net-core-7. See chapter 1 for how to get help if you have problems running the examples.

Listing 14.1 The contents of the IResponseFormatter.cs file in the Services folder

```
namespace Platform.Services {
    public interface IResponseFormatter {

        Task Format(HttpContext context, string content);
    }
}
```

The `IResponseFormatter` interface defines a single method that receives an `Http-Context` object and a `string`. To create an implementation of the interface, add a class called `TextResponseFormatter.cs` to the `Platform/Services` folder with the code shown in listing 14.2.

Listing 14.2 The contents of the TextResponseFormatter.cs file in the Services folder

```
namespace Platform.Services {
    public class TextResponseFormatter : IResponseFormatter {
        private int responseCounter = 0;

        public async Task Format(HttpContext context, string content) {
            await context.Response.
                WriteAsync($"Response {++responseCounter}:\n{content}");
        }
    }
}
```

The `TextResponseFormatter` class implements the interface and writes the content to the response as a simple string with a prefix to make it obvious when the class is used.

14.1.1 Creating a middleware component and an endpoint

Some of the examples in this chapter show how features are applied differently when using middleware and endpoints. Add a file called `WeatherMiddleware.cs` to the `Platform` folder with the code shown in listing 14.3.

Listing 14.3 The contents of the WeatherMiddleware.cs file in the Platform folder

```
namespace Platform {
    public class WeatherMiddleware {
        private RequestDelegate next;

        public WeatherMiddleware(RequestDelegate nextDelegate) {
            next = nextDelegate;
        }

        public async Task Invoke(HttpContext context) {
            if (context.Request.Path == "/middleware/class") {
                await context.Response
                 .WriteAsync("Middleware Class: It is raining in London");
            } else {
                await next(context);
            }
        }
    }
}
```

To create an endpoint that produces a similar result to the middleware component, add a file called `WeatherEndpoint.cs` to the `Platform` folder with the code shown in listing 14.4.

Listing 14.4 The contents of the WeatherEndpoint.cs file in the Platform folder

```
namespace Platform {
    public class WeatherEndpoint {

        public static async Task Endpoint(HttpContext context) {
            await context.Response
                .WriteAsync("Endpoint Class: It is cloudy in Milan");
        }
    }
}
```

14.1.2 Configuring the request pipeline

Replace the contents of the `Program.cs` file with those shown in listing 14.5. The classes defined in the previous section are applied alongside a lambda function that produce similar results.

Listing 14.5 Replacing the contents of the Program.cs file in the Platform folder

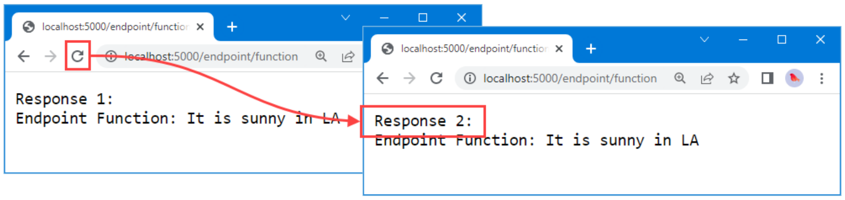
```
using Platform;
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.UseMiddleware<WeatherMiddleware>();

app.MapGet("endpoint/class", WeatherEndpoint.Endpoint);

IResponseFormatter formatter = new TextResponseFormatter();
app.MapGet("endpoint/function", async context => {
    await formatter.Format(context,
        "Endpoint Function: It is sunny in LA");
});

app.Run();
```

Start the application by opening a new PowerShell command prompt, navigating to the `Platform` project folder, and running the command shown in listing 14.6.

Listing 14.6 Starting the ASP.NET Core Runtime

```
dotnet run
```

Use a browser to request http://localhost:5000/endpoint/function, and you will see the response shown in figure 14.1. Each time you reload the browser, the counter shown in the response will be incremented.


localhost:5000/endpoint/function
localhost:5000/endpoint/function

```
Response 1:
Endpoint Function: It is sunny in LA
```

```
Response 2:
Endpoint Function: It is sunny in LA
```

Figure 14.1 Running the example application

14.2 Understanding service location and tight coupling

To understand dependency injection, it is important to start with the two problems it solves. In the sections that follow, I describe both problems addressed by dependency injection.

Taking a view on dependency injection

Dependency injection is one of the topics that readers contact me about most often. About half of the emails complain that I am "forcing" DI upon them. Oddly, the other half are complaints that I did not emphasize the benefits of DI strongly enough and other readers may not have realized how useful it can be.

Dependency injection can be a difficult topic to understand, and its value is contentious. DI can be a useful tool, but not everyone likes it—or needs it.

DI offers limited benefit if you are not doing unit testing or if you are working on a small, self-contained, and stable project. It is still helpful to understand how DI works because DI is used to access some important ASP.NET Core features described in earlier chapters, but you don't always need to embrace DI in the custom classes you write. There are alternative ways of creating shared features—two of which I describe in the following sections—and using these is perfectly acceptable if you don't like DI.

I rely on DI in my applications because I find that projects often go in unexpected directions, and being able to easily replace a component with a new implementation can save me a lot of tedious and error-prone changes. I'd rather put in some effort at the start of the project than do a complex set of edits later.

(continued)

But I am not dogmatic about dependency injection and nor should you be. Dependency injection solves a problem that doesn't arise in every project, and only you can determine whether you need DI for your project.

14.2.1 Understanding the service location problem

Most projects have features that need to be used in different parts of the application, which are known as *services*. Common examples include logging tools and configuration settings but can extend to any shared feature, including the `TextResponseFormatter` class that is defined in listing 14.2 and to handle requests by the middleware component and the lambda function.

Each `TextResponseFormatter` object maintains a counter that is included in the response sent to the browser, and if I want to incorporate the same counter into the responses generated by other endpoints, I need to have a way to make a single `TextResponseFormatter` object available in such a way that it can be easily found and consumed at every point where responses are generated.

There are many ways to make services locatable, but there are two main approaches, aside from the one that is the main topic of this chapter. The first approach is to create an object and use it as a constructor or method argument to pass it to the part of the application where it is required. The other approach is to add a `static` property to the service class that provides direct access to the shared instance, as shown in listing 14.7. This is known as the *singleton pattern*, and it was a common approach before the widespread use of dependency injection.

Listing 14.7 A singleton in the TextResponseFormatter.cs file in the Services folder

```
namespace Platform.Services {
    public class TextResponseFormatter : IResponseFormatter {
        private int responseCounter = 0;
        private static TextResponseFormatter? shared;

        public async Task Format(HttpContext context, string content) {
            await context.Response.
                WriteAsync($"Response {++responseCounter}:\n{content}");
        }

        public static TextResponseFormatter Singleton {
            get {
                if (shared == null) {
                    shared = new TextResponseFormatter();
                }
                return shared;
            }
        }
    }
}
```

This is a basic implementation of the singleton pattern, and there are many variations that pay closer attention to issues such as safe concurrent access. What's important for this chapter is that the changes in listing 14.7 rely on the consumers of the `TextResponseFormatter` service obtaining a shared object through the static `Singleton` property, as shown in listing 14.8.

Listing 14.8 Using a singleton in the WeatherEndpoint.cs file in the Platform folder

```
using Platform.Services;

namespace Platform {
    public class WeatherEndpoint {

        public static async Task Endpoint(HttpContext context) {
            await TextResponseFormatter.Singleton.Format(context,
                "Endpoint Class: It is cloudy in Milan");
        }
    }
}
```

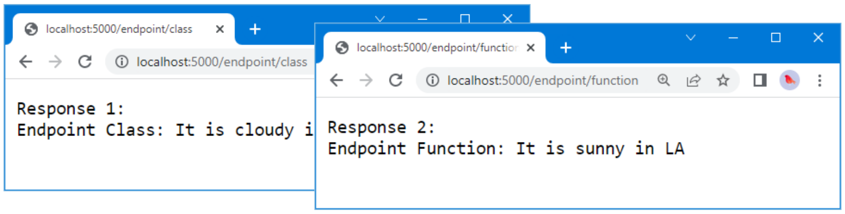
Listing 14.9 makes the same change to the lambda function in the `Program.cs` file.

Listing 14.9 Using a service in the Program.cs file in the Platform folder

```
using Platform;
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.UseMiddleware<WeatherMiddleware>();

app.MapGet("endpoint/class", WeatherEndpoint.Endpoint);

IResponseFormatter formatter = TextResponseFormatter.Singleton;
app.MapGet("endpoint/function", async context => {
    await formatter.Format(context,
        "Endpoint Function: It is sunny in LA");
});

app.Run();
```

The singleton pattern allows me to share a single `TextResponseFormatter` object so it is used by two endpoints, with the effect that a single counter is incremented by requests for two different URLs.

To see the effect of the singleton pattern, restart ASP.NET Core and request the http://localhost:5000/endpoint/class and http://localhost:5000/endpoint/function URLs. A single counter is updated for both URLs, as shown in figure 14.2.


localhost:5000/endpoint/class
localhost:5000/endpoint/class
localhost:5000/endpoint/function
localhost:5000/endpoint/function

```
Response 1:
Endpoint Class: It is cloudy i
```

```
Response 2:
Endpoint Function: It is sunny in LA
```

Figure 14.2 Implementing the singleton pattern to create a shared service

The singleton pattern is simple to understand and easy to use, but the knowledge of how services are located is spread throughout the application, and all service classes and service consumers need to understand how to access the shared object. This can lead to variations in the singleton pattern as new services are created and creates many points in the code that must be updated when there is a change. This pattern can also be rigid and doesn't allow any flexibility in how services are managed because every consumer always shares a single service object.

14.2.2 Understanding the tightly coupled components problem

Although I defined an interface in listing 14.1, the way that I have used the singleton pattern means that consumers are always aware of the implementation class they are using because that's the class whose static property is used to get the shared object. If I want to switch to a different implementation of the `IResponseFormatter` interface, I must locate every use of the service and replace the existing implementation class with the new one. There are patterns to solve this problem, too, such as the *type broker* pattern, in which a class provides access to singleton objects through their interfaces. Add a class file called `TypeBroker.cs` to the `Platform/Services` folder and use it to define the code shown in listing 14.10.

Listing 14.10 The contents of the TypeBroker.cs file in the Services folder

```
namespace Platform.Services {
    public static class TypeBroker {
        private static IResponseFormatter formatter
            = new TextResponseFormatter();

        public static IResponseFormatter Formatter => formatter;
    }
}
```

The `Formatter` property provides access to a shared service object that implements the `IResponseFormatter` interface. Consumers of the service need to know that the `TypeBroker` class is responsible for selecting the implementation that will be used, but this pattern means that service consumers can work through interfaces rather than concrete classes, as shown in listing 14.11.

Listing 14.11 Using a type broker in the WeatherEndpoint.cs file in the Platform folder

```
using Platform.Services;

namespace Platform {
    public class WeatherEndpoint {

        public static async Task Endpoint(HttpContext context) {
            await TypeBroker.Formatter.Format(context,
                "Endpoint Class: It is cloudy in Milan");
        }
    }
}
```

Listing 14.12 makes the same change to the lambda function so that both uses of the `IResponseFormatter` interface get their implementation objects from the type broker.

Listing 14.12 Using a type broker in the Program.cs file in the Platform folder

```
using Platform;
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.UseMiddleware<WeatherMiddleware>();

app.MapGet("endpoint/class", WeatherEndpoint.Endpoint);

IResponseFormatter formatter = TypeBroker.Formatter;
app.MapGet("endpoint/function", async context => {
    await formatter.Format(context,
        "Endpoint Function: It is sunny in LA");
});

app.Run();
```

This approach makes it easy to switch to a different implementation class by altering just the `TypeBroker` class and prevents service consumers from creating dependencies on a specific implementation. It also means that service classes can focus on the features they provide without having to deal with how those features will be located. To demonstrate, add a class file called `HtmlResponseFormatter.cs` to the `Platform/Services` folder with the code shown in listing 14.13.

Listing 14.13 The contents of the HtmlResponseFormatter.cs file in the Services folder

```
namespace Platform.Services {
    public class HtmlResponseFormatter : IResponseFormatter {

        public async Task Format(HttpContext context, string content) {
            context.Response.ContentType = "text/html";
```

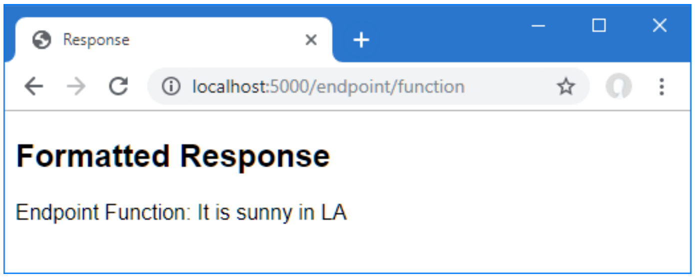
```
                await context.Response.WriteAsync($@"
                    <!DOCTYPE html>
                    <html lang=""en"">
                    <head><title>Response</title></head>
                    <body>
                        <h2>Formatted Response</h2>
                        <span>{content}</span>
                    </body>
                    </html>");
            }
        }
    }
```

This implementation of the `IResponseFormatter` sets the `ContentType` property of the `HttpResponse` object and inserts the content into an HTML template string. To use the new formatter class, I only need to change the `TypeBroker`, as shown in listing 14.14.

Listing 14.14 changing the TypeBroker.cs file in the Platform/Services folder

```
namespace Platform.Services {
    public static class TypeBroker {
        private static IResponseFormatter formatter
            = new HtmlResponseFormatter();

        public static IResponseFormatter Formatter => formatter;
    }
}
```

To confirm the new formatter works, restart ASP.NET Core and request http://localhost:5000/endpoint/function, which will produce the result shown in figure 14.3.



Formatted Response

Endpoint Function: It is sunny in LA

Figure 14.3 Using a different service implementation class

14.3 Using dependency injection

Dependency injection provides an alternative approach to providing services that tidy up the rough edges that arise in the singleton and type broker patterns, and is integrated with other ASP.NET Core features. Listing 14.15 shows the use of ASP.NET Core dependency injection to replace the type broker from the previous section.

Listing 14.15 Using dependency injection in the Program.cs file in the Platform folder

```
using Platform;
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton<IResponseFormatter,
    HtmlResponseFormatter>();

var app = builder.Build();

app.UseMiddleware<WeatherMiddleware>();

app.MapGet("endpoint/class", WeatherEndpoint.Endpoint);

//IResponseFormatter formatter = TypeBroker.Formatter;
app.MapGet("endpoint/function",
    async (HttpContext context, IResponseFormatter formatter) => {
        await formatter.Format(context,
            "Endpoint Function: It is sunny in LA");
});

app.Run();
```

Services are registered using extension methods defined by the `IServiceCollection` interface, an implementation of which is obtained using the `WebApplication-Builder.Services` property. In the listing, I used an extension method to create a service for the `IResponseFormatter` interface:

```
...
builder.Services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>();
...
```

The `AddSingleton` method is one of the extension methods available for services and tells ASP.NET Core that a single object should be used to satisfy all demands for the service (the other extension methods are described in the "Using Service Lifecycles" section). The interface and the implementation class are specified as generic type arguments. To consume the service, I added parameters to the functions that handle requests, like this:

```
...
async (HttpContext context, IResponseFormatter formatter) => {
...
```

Many of the methods that are used to register middleware or create endpoints will accept any function, which allows parameters to be defined for the services that are required to produce a response. One consequence of this feature is that the C# compiler can't determine the parameter types, which is why I had to specify them in the listing.

The new parameter declares a dependency on the `IResponseFormatter` interface, and the function is said to depend on the interface. Before the function is invoked to handle a request, its parameters are inspected, the dependency is detected, and the application's services are inspected to determine whether it is possible to resolve the dependency.

The call to the AddSingleton method told the dependency injection system that a dependency on the IResponseFormatter interface can be resolved with an HtmlResponseFormatter object. The object is created and used as an argument to invoke the handler function. Because the object that resolves the dependency is provided from outside the function that uses it, it is said to have been *injected*, which is why the process is known as *dependency injection*.

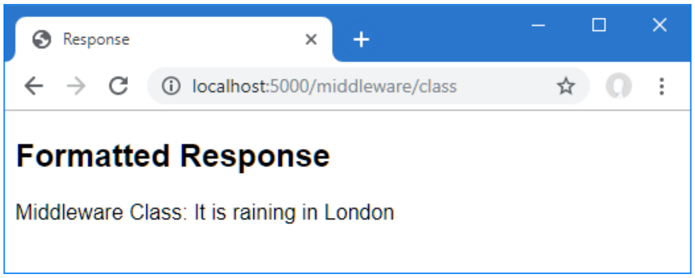
14.3.1 Using a Service with a Constructor Dependency

Defining a service and consuming it in the same code file may not seem impressive, but once a service is defined, it can be used almost anywhere in an ASP.NET Core application. Listing 14.16 declares a dependency on the IResponseFormatter interface in the middleware class defined at the start of the chapter.

Listing 14.16 A dependency in the WeatherMiddleware.cs file in the Platform folder

```
using Platform.Services;

namespace Platform {
    public class WeatherMiddleware {
        private RequestDelegate next;
        private IResponseFormatter formatter;

        public WeatherMiddleware(RequestDelegate nextDelegate,
                IResponseFormatter respFormatter) {
            next = nextDelegate;
            formatter = respFormatter;
        }

        public async Task Invoke(HttpContext context) {
            if (context.Request.Path == "/middleware/class") {
                await formatter.Format(context,
                    "Middleware Class: It is raining in London");
            } else {
                await next(context);
            }
        }
    }
}
```

To declare the dependency, I added a constructor parameter. To see the result, restart ASP.NET Core and request the http://localhost:5000/middleware/class URL, which will produce the response shown in figure 14.4.



Formatted Response

Middleware Class: It is raining in London

Figure 14.4 Declaring a dependency in a middleware class

When the request pipeline is being set up, the ASP.NET Core platform reaches the statement in the `Program.cs` file that adds the `WeatherMiddleware` class as a component.

```
...
app.UseMiddleware<WeatherMiddleware>();
...
```

The platform understands it needs to create an instance of the `WeatherMiddleware` class and inspects the constructor. The dependency on the `IResponseFormatter` interface is detected, the services are inspected to see if the dependency can be resolved, and the shared service object is used when the constructor is invoked.

There are two important points to understand about this example. The first is that `WeatherMiddleware` doesn't know which implementation class will be used to resolve its dependency on the `IResponseFormatter` interface—it just knows that it will receive an object that conforms to the interface through its constructor parameter. Second, the `WeatherMiddleware` class doesn't know how the dependency is resolved—it just declares a constructor parameter and relies on ASP.NET Core to figure out the details. This is a more elegant approach than my implementations of the singleton and type broker patterns earlier in the chapter, and I can change the implementation class used to resolve the service by changing the generic type parameters used in the `Program.cs` file.

14.3.2 Getting services from the HttpContext object

ASP.NET Core does a good job of supporting dependency injection as widely as possible but there will be times when you are not working directly with the ASP.NET Core API and won't have a way to declare your service dependencies directly.

Services can be accessed through the `HttpContext` object, which is used to represent the current request and response, as shown in listing 14.17. You may find that you receive an `HttpContext` object even if you are working with third-party code that acts as an intermediary to ASP.NET Core and which doesn't allow you to resolve services.

NOTE This example is a demonstration of a common problem but ASP.NET Core does support dependency injection for endpoint delegates.

Listing 14.17 Using the HttpContext in the WeatherEndpoint.cs file in the Platform folder

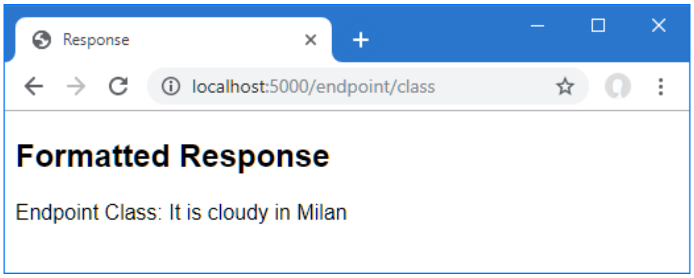
```
using Platform.Services;

namespace Platform {
    public class WeatherEndpoint {

        public static async Task Endpoint(HttpContext context) {
            IResponseFormatter formatter = context.RequestServices
                .GetRequiredService<IResponseFormatter>();
            await formatter.Format(context,
                "Endpoint Class: It is cloudy in Milan");
        }
    }
}
```

The `HttpContext.RequestServices` property returns an object that implements the `IServiceProvider` interfaces, which provides access to the services that have been configured in the `Program.cs` file. The `Microsoft.Extensions.DependencyInjection` namespace contains extension methods for the `IServiceProvider` interface that allow individual services to be obtained, as described in table 14.3.

Table 14.3 The IServiceProvider extension methods for obtaining services

Name	Description
`GetService<T>()`	This method returns a service for the type specified by the generic type parameter or `null` if no such service has been defined.
`GetService(type)`	This method returns a service for the type specified or `null` if no such service has been defined.
`GetRequiredService<T>()`	This method returns a service specified by the generic type parameter and throws an exception if a service isn't available.
`GetRequiredService(type)`	This method returns a service for the type specified and throws an exception if a service isn't available.

When the `Endpoint` method is invoked in listing 14.17, the `GetRequiredService<T>` method is used to obtain an `IResponseFormatter` object, which is used to format the response. To see the effect, restart ASP.NET Core and use the browser to request http://localhost:5000/endpoint/class, which will produce the formatted response shown in figure 14.5.



Formatted Response

Endpoint Class: It is cloudy in Milan

Figure 14.5 Using a service in an endpoint class

Using the activation utility class

I defined static methods for endpoint classes in chapter 13 because it makes them easier to use when creating routes. But for endpoints that require services, it can often be easier to use a class that can be instantiated because it allows for a more generalized approach to handling services. Listing 14.18 revises the endpoint with a constructor and removes the `static` keyword from the `Endpoint` method.

Listing 14.18 Revising the endpoint in the WeatherEndpoint.cs file in the Platform folder

```
using Platform.Services;

namespace Platform {
    public class WeatherEndpoint {
        private IResponseFormatter formatter;

        public WeatherEndpoint(IResponseFormatter responseFormatter) {
            formatter = responseFormatter;
        }

        public async Task Endpoint(HttpContext context) {
            await formatter.Format(context,
                "Endpoint Class: It is cloudy in Milan");
        }
    }
}
```

The most common use of dependency injection in ASP.NET Core applications is in class constructors. Injection through methods, such as performed for middleware classes, is a complex process to re-create, but there are some useful built-in tools that take care of inspecting constructors and resolving dependencies using services. Create a file called `EndpointExtensions.cs` to the `Services` folder with the content shown in listing 14.19.

Listing 14.19 The contents of the EndpointExtensions.cs file in the Services folder

```
using System.Reflection;

namespace Microsoft.AspNetCore.Builder {
    public static class EndpointExtensions {

        public static void MapEndpoint<T>(this IEndpointRouteBuilder app,
                string path, string methodName = "Endpoint") {

            MethodInfo? methodInfo = typeof(T).GetMethod(methodName);
            if (methodInfo?.ReturnType != typeof(Task)) {
                throw new System.Exception("Method cannot be used");
            }
            T endpointInstance =
                ActivatorUtilities.CreateInstance<T>(app.ServiceProvider);
            app.MapGet(path, (RequestDelegate)methodInfo
                .CreateDelegate(typeof(RequestDelegate),
                    endpointInstance));
        }
    }
}
```

The `MapEndpoint` extension method accepts a generic type parameter that specifies the endpoint class that will be used. The other arguments are the path that will be used to create the route and the name of the endpoint class method that processes requests.

A new instance of the endpoint class is created, and a delegate to the specified method is used to create a route. Like any code that uses .NET reflection, the extension method in listing 14.19 can be difficult to read, but this is the key statement for this chapter:

```
...
T endpointInstance =
    ActivatorUtilities.CreateInstance<T>(app.ServiceProvider);
...
```

The `ActivatorUtilities` class, defined in the `Microsoft.Extensions.Dependency-Injection` namespace, provides methods for instantiating classes that have dependencies declared through their constructor. Table 14.4 shows the most useful `ActivatorUtilities` methods.

Table 14.4 The ActivatorUtilities methods

Name	Description
`CreateInstance<T>(services, args)`	This method creates a new instance of the class specified by the type parameter, resolving dependencies using the services and additional (optional) arguments.
`CreateInstance(services, type, args)`	This method creates a new instance of the class specified by the parameter, resolving dependencies using the services and additional (optional) arguments.
`GetServiceOrCreateInstance<T>(services, args)`	This method returns a service of the specified type, if one is available, or creates a new instance if there is no service.
`GetServiceOrCreateInstance(services, type, args)`	This method returns a service of the specified type, if one is available, or creates a new instance if there is no service.

These methods make it easy to instantiate classes that declare constructor dependencies. Both methods resolve constructor dependencies using services through an `IServiceProvider` object and an optional array of arguments that are used for dependencies that are not services. These methods make it easy to apply dependency injection to custom classes, and the use of the `CreateInstance` method results in an extension method that can create routes with endpoint classes that consume services. Listing 14.20 uses the new extension method to create a route.

Listing 14.20 Creating a route in the Program.cs file in the Platform folder

```
using Platform;
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton<IResponseFormatter,
    HtmlResponseFormatter>();

var app = builder.Build();

app.UseMiddleware<WeatherMiddleware>();

//app.MapGet("endpoint/class", WeatherEndpoint.Endpoint);
app.MapEndpoint<WeatherEndpoint>("endpoint/class");

app.MapGet("endpoint/function",
    async (HttpContext context, IResponseFormatter formatter) => {
        await formatter.Format(context,
            "Endpoint Function: It is sunny in LA");
});

app.Run();
```

To confirm that requests are routed to the endpoint, restart ASP.NET Core and request the http://localhost:5000/endpoint/class URL, which should produce the same response as shown in figure 14.5.

14.4 Using Service Lifecycles

When I created the service in the previous section, I used the `AddSingleton` extension method, like this:

```
...
builder.Services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>();
...
```

The `AddSingleton` method produces a service that is instantiated the first time it is used to resolve a dependency and is then reused for each subsequent dependency. This means that any dependency on the `IResponseFormatter` object will be resolved using the same `HtmlResponseFormatter` object.

Singletons are a good way to get started with services, but there are some problems for which they are not suited, so ASP.NET Core supports *scoped* and *transient* services, which give different lifecycles for the objects that are created to resolve dependencies. Table 14.5 describes the set of methods used to create services. There are versions of these methods that accept types as conventional arguments, as demonstrated in the "Using Unbound Types in Services" section, later in this chapter.

Table 14.5. The extension methods for creating services

Name	Description
`AddSingleton<T, U>()`	This method creates a single object of type `U` that is used to resolve all dependencies on type `T`.
`AddTransient<T, U>()`	This method creates a new object of type `U` to resolve each dependency on type `T`.
`AddScoped<T, U>()`	This method creates a new object of type `U` that is used to resolve dependencies on `T` within a single scope, such as request.

There are versions of the methods in Table 14.5 that have a single type argument, which allows a service to be created that solves the service location problem without addressing the tightly coupled issue. You can see an example of this type of service in chapter 24, where I share a simple data source that isn't accessed through an interface.

14.4.1 Creating transient services

The `AddTransient` method does the opposite of the `AddSingleton` method and creates a new instance of the implementation class for every dependency that is resolved. To create a service that will demonstrate the use of service lifecycles, add a file called `GuidService.cs` to the `Platform/Services` folder with the code shown in listing 14.21.

Listing 14.21 The contents of the GuidService.cs file in the Services folder

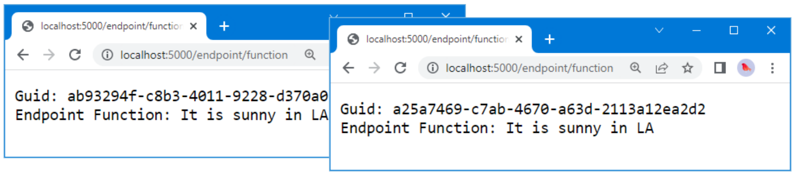
```
namespace Platform.Services {

    public class GuidService : IResponseFormatter {
        private Guid guid = Guid.NewGuid();

        public async Task Format(HttpContext context, string content) {
            await context.Response.WriteAsync($"Guid: {guid}\n{content}");
        }
    }
}
```

The `Guid` struct generates a unique identifier, which will make it obvious when a different instance is used to resolve a dependency on the `IResponseFormatter` interface. In listing 14.22, I have changed the statement that creates the `IResponseFormatter` service to use the `AddTransient` method and the `GuidService` implementation class.

Listing 14.22 Creating a transient service in the Program.cs file in the Platform folder

```
using Platform;
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddTransient<IResponseFormatter, GuidService>();

var app = builder.Build();

app.UseMiddleware<WeatherMiddleware>();

app.MapEndpoint<WeatherEndpoint>("endpoint/class");

app.MapGet("endpoint/function",
    async (HttpContext context, IResponseFormatter formatter) => {
        await formatter.Format(context,
            "Endpoint Function: It is sunny in LA");
});

app.Run();
```

If you restart ASP.NET Core and request the http://localhost:5000/endpoint/function URL, you will receive the responses similar to the ones shown in figure 14.6. Each response will be shown with a different GUID value, confirming that transient service objects have been used to resolve the dependency on the `IResponseFormatter` service.



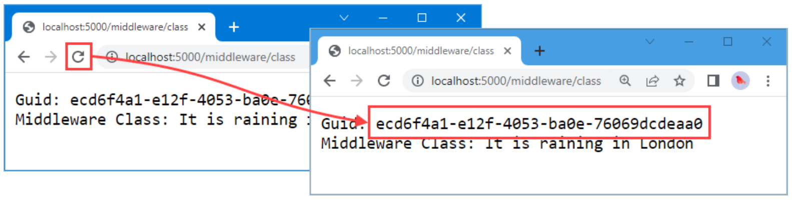
```
Guid: ab93294f-c8b3-4011-9228-d370a0
Endpoint Function: It is sunny in LA
```

```
Guid: a25a7469-c7ab-4670-a63d-2113a12ea2d2
Endpoint Function: It is sunny in LA
```

Figure 14.6 Using transient services

14.4.2 Avoiding the transient service reuse pitfall

There is a pitfall when using transient services, which you can see by requesting http://localhost:5000/middleware/class and clicking the reload button. Unlike the previous example, the same GUID value is shown in every response, as shown in figure 14.7.



```
Guid: ecd6f4a1-e12f-4053-ba0e-76
Middleware Class: It is raining
```

```
Guid: ecd6f4a1-e12f-4053-ba0e-76069dcdeaa0
Middleware Class: It is raining in London
```

Figure 14.7 The same GUID values appearing in responses

New service objects are created only when dependencies are resolved, not when services are used. The components and endpoints in the example application have their dependencies resolved only when the application starts and the top-level statements in the `Program.cs` file are executed. Each receives a separate service object, which is then reused for every request that is processed.

To address this issue, I have to ensure that the dependency is resolved every time the `Invoke` method is called, as shown in listing 14.23.

Listing 14.23 Moving dependencies in the Platform/WeatherMiddleware.cs file

```
using Platform.Services;

namespace Platform {
    public class WeatherMiddleware {
        private RequestDelegate next;
        //private IResponseFormatter formatter;

        public WeatherMiddleware(RequestDelegate nextDelegate) {
            next = nextDelegate;
            //formatter = respFormatter;
        }

        public async Task Invoke(HttpContext context,
                IResponseFormatter formatter) {
            if (context.Request.Path == "/middleware/class") {
                await formatter.Format(context,
                    "Middleware Class: It is raining in London");
            } else {
                await next(context);
            }
        }
    }
}
```

The ASP.NET Core platform will resolve dependencies declared by the `Invoke` method every time a request is processed, which ensures that a new transient service object is created.

The `ActivatorUtilities` class doesn't deal with resolving dependencies for methods. The simplest way of solving this issue for endpoints is to explicitly request services when each request is handled, which is the approach I used earlier when showing how services are used. It is also possible to enhance the extension method to request services on behalf of an endpoint, as shown in listing 14.24.

Listing 14.24 Requesting services in the Services/EndpointExtensions.cs file

```
using System.Reflection;

namespace Microsoft.AspNetCore.Builder {

    public static class EndpointExtensions {

        public static void MapEndpoint<T>(this IEndpointRouteBuilder app,
                string path, string methodName = "Endpoint") {

            MethodInfo? methodInfo = typeof(T).GetMethod(methodName);
            if (methodInfo?.ReturnType != typeof(Task)) {
                throw new System.Exception("Method cannot be used");
            }
            T endpointInstance =
                ActivatorUtilities.CreateInstance<T>(app.ServiceProvider);

            ParameterInfo[] methodParams = methodInfo!.GetParameters();

            app.MapGet(path, context =>
                (Task)methodInfo.Invoke(endpointInstance,
                  methodParams.Select(p =>
                    p.ParameterType == typeof(HttpContext) ? context
                      : app.ServiceProvider.GetService(p.ParameterType))
                    .ToArray())!);
        }
    }
}
```

The code in listing 14.24 isn't as efficient as the approach taken by the ASP.NET Core platform for middleware components. All the parameters defined by the method that handles requests are treated as services to be resolved, except for the `HttpContext` parameter. A route is created with a delegate that resolves the services for every request and invokes the method that handles the request. Listing 14.25 revises the `WeatherEndpoint` class to move the dependency on `IResponseFormatter` to the `Endpoint` method so that a new service object will be received for every request.

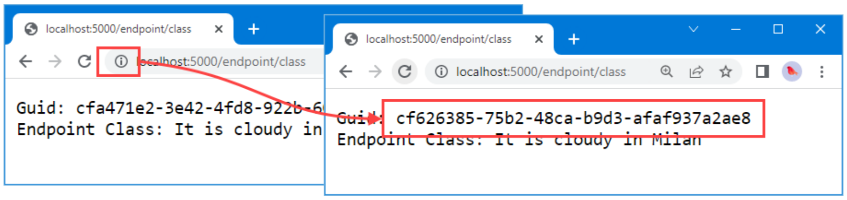
Listing 14.25 Moving the dependency in the Platform/WeatherEndpoint.cs file

```
using Platform.Services;

namespace Platform {
    public class WeatherEndpoint {
        //private IResponseFormatter formatter;

        //public WeatherEndpoint(IResponseFormatter responseFormatter) {
        //    formatter = responseFormatter;
        //}

        public async Task Endpoint(HttpContext context,
                IResponseFormatter formatter) {
            await formatter.Format(context,
                "Endpoint Class: It is cloudy in Milan");
        }
    }
}
```

The changes in listing 14.23 to listing 14.25 ensure that the transient service is resolved for every request, which means that a new `GuidService` object is created and every response contains a unique ID.

Restart ASP.NET Core, navigate to http://localhost:5000/endpoint/class, and click the browser's reload button. Each time you reload, a new request is sent to ASP.NET

Core, and the component or endpoint that handles the request receives a new service object, such that a different GUID is shown in each response, as shown in figure 14.8.



```
Guid: cfa471e2-3e42-4fd8-922b-6
Endpoint Class: It is cloudy in
```

```
Guid: cf626385-75b2-48ca-b9d3-afaf937a2ae8
Endpoint Class: It is cloudy in Milan
```

Figure 14.8 Using a transient service

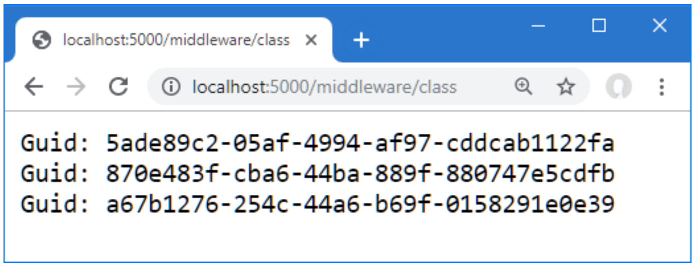
14.4.3 *Using scoped services*

Scoped services strike a balance between singleton and transient services. Within a scope, dependencies are resolved with the same object. A new scope is started for each HTTP request, which means that a service object will be shared by all the components that handle that request. To prepare for a scoped service, listing 14.26 changes the `WeatherMiddleware` class to declare three dependencies on the same service.

Listing 14.26 Adding dependencies in the Platform/WeatherMiddleware.cs file

```
using Platform.Services;

namespace Platform {
    public class WeatherMiddleware {
        private RequestDelegate next;

        public WeatherMiddleware(RequestDelegate nextDelegate) {
            next = nextDelegate;
        }

        public async Task Invoke(HttpContext context,
                IResponseFormatter formatter1,
                IResponseFormatter formatter2,
                IResponseFormatter formatter3) {
            if (context.Request.Path == "/middleware/class") {
                await formatter1.Format(context, string.Empty);
                await formatter2.Format(context, string.Empty);
                await formatter3.Format(context, string.Empty);
            } else {
                await next(context);
            }
        }
    }
}
```

Declaring several dependencies on the same service isn't required in real projects, but it is useful for this example because each dependency is resolved independently. Since the `IResponseFormatter` service was created with the `AddTransient` method, each dependency is resolved with a different object. Restart ASP.NET Core and request http://localhost:5000/middleware/class, and you will see that a different GUID is used for each of the three messages written to the response, as shown in figure 14.9. When you reload the browser, a new set of three GUIDs is displayed.


localhost:5000/middleware/class
localhost:5000/middleware/class

```
Guid: 5ade89c2-05af-4994-af97-cddcab1122fa
Guid: 870e483f-cba6-44ba-889f-880747e5cdfb
Guid: a67b1276-254c-44a6-b69f-0158291e0e39
```

Figure 14.9 Resolving dependencies on a transient service

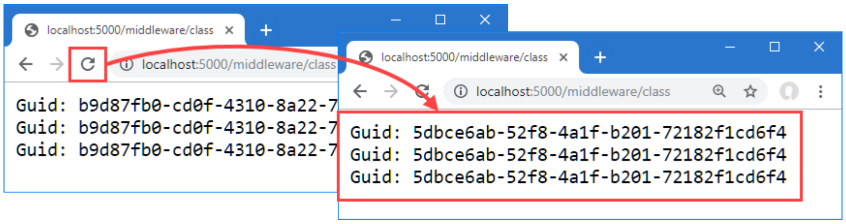
Listing 14.27 changes the `IResponseFormatter` service to use the scoped lifecycle with the `AddScoped` method.

> **TIP** You can create scopes through the `CreateScope` extension method for the `IServiceProvider` interface. The result is an `IServiceProvider` that is associated with a new scope and that will have its own implementation objects for scoped services.

Listing 14.27 Using a scoped service in the Program.cs file in the Platform folder

```
using Platform;
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IResponseFormatter, GuidService>();

var app = builder.Build();

app.UseMiddleware<WeatherMiddleware>();

app.MapEndpoint<WeatherEndpoint>("endpoint/class");

app.MapGet("endpoint/function",
    async (HttpContext context, IResponseFormatter formatter) => {
        await formatter.Format(context,
            "Endpoint Function: It is sunny in LA");
});

app.Run();
```

Restart ASP.NET Core and request http://localhost:5000/middleware/class again, and you will see that the same GUID is used to resolve all three dependencies declared by the middleware component, as shown in figure 14.10. When the browser is reloaded, the HTTP request sent to ASP.NET Core creates a new scope and a new service object.


localhost:5000/middleware/class
localhost:5000/middleware/class

```
Guid: b9d87fb0-cd0f-4310-8a22-7
Guid: b9d87fb0-cd0f-4310-8a22-7
Guid: b9d87fb0-cd0f-4310-8a22-7
```

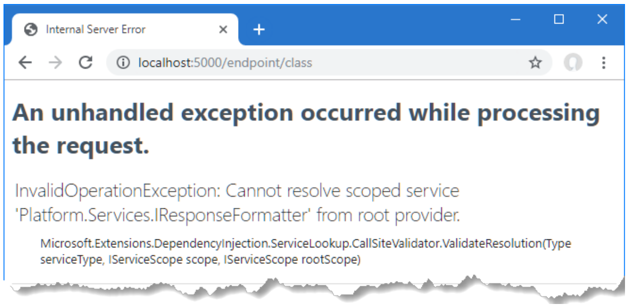
```
Guid: 5dbce6ab-52f8-4a1f-b201-72182f1cd6f4
Guid: 5dbce6ab-52f8-4a1f-b201-72182f1cd6f4
Guid: 5dbce6ab-52f8-4a1f-b201-72182f1cd6f4
```

Figure 14.10 Using a scoped service

Avoiding the scoped service validation pitfall

Service consumers are unaware of the lifecycle that has been selected for singleton and transient services: they declare a dependency or request a service and get the object they require.

Scoped services can be used only within a scope. A new scope is created automatically for each request that was received. Requesting a scoped service outside of a scope causes an exception. To see the problem, request http://localhost:5000/endpoint/class, which will generate the exception response shown in figure 14.11.



An unhandled exception occurred while processing the request.

InvalidOperationException: Cannot resolve scoped service 'Platform.Services.IResponseFormatter' from root provider.

Microsoft.Extensions.DependencyInjection.ServiceLookup.CallSiteValidator.ValidateResolution(Type serviceType, IServiceScope scope, IServiceScope rootScope)

Figure 14.11 Requesting a scoped service

The extension method that configures the endpoint resolves services through an `IServiceProvider` object obtained from the routing middleware, like this:

```
...
app.ServiceProvider.GetService(p.ParameterType))
...
```

Accessing scoped services through the context object

The `HttpContext` class defines a `RequestServices` property that returns an `IServiceProvider` object that allows access to scoped services, as well as singleton and transient services. This fits well with the most common use of scoped services, which is to use a single service object for each HTTP request. Listing 14.28 revises the endpoint extension method so that dependencies are resolved using the services provided through the `HttpContext`.

Listing 14.28 Using a service in the EndpointExtensions.cs file in the Services folder

```
using System.Reflection;

namespace Microsoft.AspNetCore.Builder {

    public static class EndpointExtensions {

        public static void MapEndpoint<T>(this IEndpointRouteBuilder app,
                string path, string methodName = "Endpoint") {

            MethodInfo? methodInfo = typeof(T).GetMethod(methodName);
            if (methodInfo?.ReturnType != typeof(Task)) {
                throw new System.Exception("Method cannot be used");
            }
            T endpointInstance =
                ActivatorUtilities.CreateInstance<T>(app.ServiceProvider);

            ParameterInfo[] methodParams = methodInfo!.GetParameters();

            app.MapGet(path, context =>
                (Task)methodInfo.Invoke(endpointInstance,
                  methodParams.Select(p =>
                    p.ParameterType == typeof(HttpContext) ? context
                      : context.RequestServices
                        .GetService(p.ParameterType))
                    .ToArray())!);
        }
    }
}
```

Using the `HttpContext.RequestServices` property ensures that services are resolved within the scope of the current HTTP request, which ensures that endpoints don't use scoped services inappropriately.

Creating new handlers for each request

Notice that the `ActivatorUtilities.CreateInstance<T>` method is still used to create an instance of the endpoint class in listing 14.28.

This presents a problem because it requires endpoint classes to know the lifecycles of the services on which they depend. The `WeatherEndpoint` class depends on the `IResponseFormatter` service and must know that a dependency can be declared only through the `Endpoint` method and not the constructor.

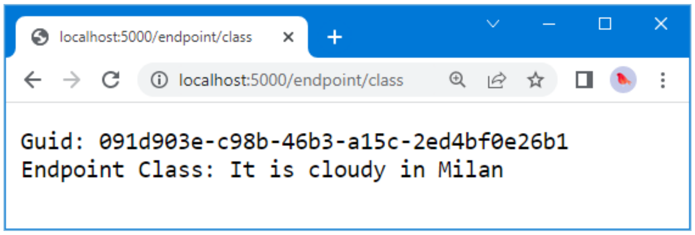
To remove the need for this knowledge, a new instance of the endpoint class can be created to handle each request, as shown in listing 14.29, which allows constructor and method dependencies to be resolved without needing to know which services are scoped.

Listing 14.29 Instantiating in the EndpointExtensions.cs file in the Services folder

```
using System.Reflection;

namespace Microsoft.AspNetCore.Builder {
    public static class EndpointExtensions {

        public static void MapEndpoint<T>(this IEndpointRouteBuilder app,
                string path, string methodName = "Endpoint") {

            MethodInfo? methodInfo = typeof(T).GetMethod(methodName);
            if (methodInfo?.ReturnType != typeof(Task)) {
                throw new System.Exception("Method cannot be used");
            }
            //T endpointInstance =
            //ActivatorUtilities.CreateInstance<T> (app.ServiceProvider);

            ParameterInfo[] methodParams = methodInfo!.GetParameters();

            app.MapGet(path, context => {
                T endpointInstance =
                    ActivatorUtilities.CreateInstance<T>
                        (context.RequestServices);
                return (Task)methodInfo.Invoke(endpointInstance!,
                    methodParams.Select(p =>
                    p.ParameterType == typeof(HttpContext)
                    ? context
                    : context.RequestServices.GetService(p.ParameterType))
                        .ToArray())!;
            });
        }
    }
}
```

This approach requires a new instance of the endpoint class to handle each request, but it ensures that no knowledge of service lifecycles is required.

Restart ASP.NET Core and request http://localhost:5000/endpoint/class. The scoped service will be obtained from the context, producing the responses shown in figure 14.12.



```
Guid: 091d903e-c98b-46b3-a15c-2ed4bf0e26b1
Endpoint Class: It is cloudy in Milan
```

Figure 14.12 Using scoped services in lambda functions

Accessing scoped services when configuring the request pipeline

If you require a scoped service to configure the application in the `Program.cs` file, then you can create a new scope and then request the service, like this:

(continued)

```
...
app.Services.CreateScope().ServiceProvider
    .GetRequiredService<MyScopedService>();
...
```

The `CreateScope` method creates a scope that allows scoped services to be accessed. If you try to obtain a scoped service without creating a scope, then you will receive an exception.

14.5 Other dependency injection features

In the sections that follow, I describe some additional features available when using dependency injection. These are not required for all projects, but they are worth understanding because they provide context for how dependency injection works and can be helpful when the standard features are not quite what a project requires.

14.5.1 Creating dependency chains

When a class is instantiated to resolve a service dependency, its constructor is inspected, and any dependencies on services are resolved. This allows one service to declare a dependency on another service, creating a chain that is resolved automatically. To demonstrate, add a class file called `TimeStamping.cs` to the `Platform/Services` folder with the code shown in listing 14.30.

Listing 14.30 The contents of the TimeStamping.cs file in the Services folder

```
namespace Platform.Services {

    public interface ITimeStamper {
        string TimeStamp { get; }
    }

    public class DefaultTimeStamper : ITimeStamper {

        public string TimeStamp {
            get => DateTime.Now.ToShortTimeString();
        }
    }
}
```

The class file defines an interface named `ITimeStamper` and an implementation class named `DefaultTimeStamper`. Next, add a file called `TimeResponseFormatter.cs` to the `Platform/Services` folder with the code shown in listing 14.31.

Listing 14.31 The contents of the TimeResponseFormatter.cs file in the Services folder

```
namespace Platform.Services {
    public class TimeResponseFormatter : IResponseFormatter {
        private ITimeStamper stamper;
        public TimeResponseFormatter(ITimeStamper timeStamper) {
            stamper = timeStamper;
        }

        public async Task Format(HttpContext context, string content) {
            await context.Response.WriteAsync($"{stamper.TimeStamp}: "
                + content);
        }
    }
}
```

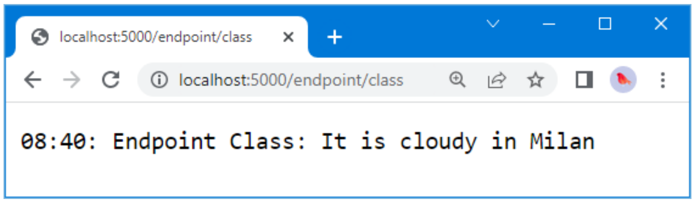
The `TimeResponseFormatter` class is an implementation of the `IResponseFormatter` interface that declares a dependency on the `ITimeStamper` interface with a constructor parameter. Listing 14.32 defines services for both interfaces in the `Program.cs` file.

NOTE Services don't need to have the same lifecycle as their dependencies, but you can end up with odd effects if you mix lifecycles. Lifecycles are applied only when a dependency is resolved, which means that if a scoped service depends on a transient service, for example, then the transient object will behave as though it was assigned the scoped lifecycle.

Listing 14.32 Configuring services in the Program.cs file in the Platform folder

```
using Platform;
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IResponseFormatter, TimeResponseFormatter>();
builder.Services.AddScoped<ITimeStamper, DefaultTimeStamper>();

var app = builder.Build();

app.UseMiddleware<WeatherMiddleware>();

app.MapEndpoint<WeatherEndpoint>("endpoint/class");

app.MapGet("endpoint/function",
    async (HttpContext context, IResponseFormatter formatter) => {
        await formatter.Format(context,
            "Endpoint Function: It is sunny in LA");
});

app.Run();
```

When a dependency on the `IResponseFormatter` service is resolved, the `TimeResponseFormatter` constructor will be inspected, and its dependency on the `ITimeStamper` service will be detected. A `DefaultTimeStamper` object will be created and injected into the `TimeResponseFormatter` constructor, which allows the original dependency to be resolved. To see the dependency chain in action, restart ASP.NET Core and request http://localhost:5000/endpoint/class, and you will see the timestamp generated by the `DefaultTimeStamper` class included in the response produced by the `TimeResponseFormatter` class, as shown in figure 14.13.



```
08:40: Endpoint Class: It is cloudy in Milan
```

Figure 14.13 Creating a chain of dependencies

14.5.2 Accessing services in the Program.cs file

A common requirement is to use the application's configuration settings to alter the set of services that are created in the `Program.cs` file. This presents a problem because the configuration is presented as a service and services cannot normally be accessed until after the `WebApplicationBuilder.Build` method is invoked.

To address this issue, the `WebApplication` and `WebApplicationBuilder` classes define properties that provide access to the built-in services that provide access to the application configuration, as described in table 14.6.

Table 14.6 The WebApplication and WebApplicationBuilder properties for configuration services

Name	Description
`IConfiguration`	This property returns an implementation of the `IConfiguration` interface, which provides access to the application's configuration settings, as described in chapter 15. `Configuration`.
`Environment`	This property returns an implementation of the `IWebHostEnvironment` interface, which provides information about the environment in which the application is being executed and whose principal use is to determine if the application is configured for development or deployment, as described in chapter 15.

These services are described in chapter 15, but what's important for this chapter is that they can be used to customize which services are configured in the `Program.cs` file, as shown in listing 14.33.

Listing 14.33 Accessing configuration data in the Program.cs file in the Platform folder

```
using Platform;
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

IWebHostEnvironment env = builder.Environment;
```

```
if (env.IsDevelopment()) {
    builder.Services.AddScoped<IResponseFormatter,
        TimeResponseFormatter>();
    builder.Services.AddScoped<ITimeStamper, DefaultTimeStamper>();
} else {
    builder.Services.AddScoped<IResponseFormatter,
        HtmlResponseFormatter>();
}

var app = builder.Build();

app.UseMiddleware<WeatherMiddleware>();

app.MapEndpoint<WeatherEndpoint>("endpoint/class");

app.MapGet("endpoint/function",
    async (HttpContext context, IResponseFormatter formatter) => {
        await formatter.Format(context,
            "Endpoint Function: It is sunny in LA");
});

app.Run();
```

This example uses the `Environment` property to get an implementation of the `IWebHostEnvironment` interface and uses its `IsDevelopment` extension method to decide which services are set up for the application.

14.5.3 Using service factory functions

Factory functions allow you to take control of the way that service implementation objects are created, rather than relying on ASP.NET Core to create instances for you. There are factory versions of the `AddSingleton`, `AddTransient`, and `AddScoped` methods, all of which are used with a function that receives an `IServiceProvider` object and returns an implementation object for the service.

One use for factory functions is to define the implementation class for a service as a configuration setting, which is read through the `IConfguration` service. This requires the `WebApplicationBuilder` properties described in the previous section. Listing 14.34 adds a factory function for the `IResponseFormatter` service that gets the implementation class from the configuration data.

Listing 14.34 Using a factory function in the Program.cs file in the Platform folder

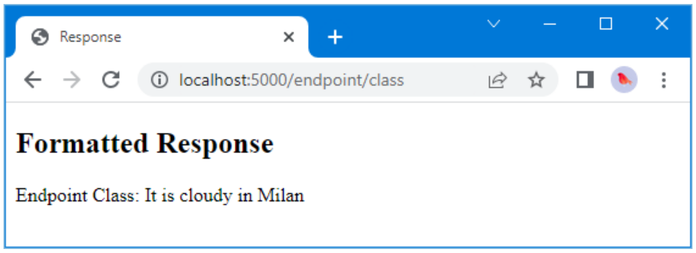
```
using Platform;
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

//IWebHostEnvironment env = builder.Environment;
IConfiguration config = builder.Configuration;

builder.Services.AddScoped<IResponseFormatter>(serviceProvider => {
    string? typeName = config["services:IResponseFormatter"];
    return (IResponseFormatter)ActivatorUtilities
        .CreateInstance(serviceProvider, typeName == null
            ? typeof(GuidService) : Type.GetType(typeName, true)!);
});
builder.Services.AddScoped<ITimeStamper, DefaultTimeStamper>();

var app = builder.Build();

app.UseMiddleware<WeatherMiddleware>();

app.MapEndpoint<WeatherEndpoint>("endpoint/class");

app.MapGet("endpoint/function",
    async (HttpContext context, IResponseFormatter formatter) => {
        await formatter.Format(context,
            "Endpoint Function: It is sunny in LA");
});

app.Run();
```

The factory function reads a value from the configuration data, which is converted into a type and passed to the `ActivatorUtilities.CreateInstance` method. Listing 14.35 adds a configuration setting to the `appsettings.Development.json` file that selects the `HtmlResponseFormatter` class as the implementation for the `IResponse-Formatter` service. The JSON configuration file is described in detail in chapter 15.

Listing 14.35 A setting in the appsettings.Development.json file in the Platform folder

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "services": {
    "IResponseFormatter": "Platform.Services.HtmlResponseFormatter"
  }
}
```

When a dependency on the `IResponseFormatter` service is resolved, the factory function creates an instance of the type specified in the configuration file. Restart ASP.NET Core and request the http://localhost:5000/endpoint/class URL, which will produce the response shown in figure 14.14.



Formatted Response

Endpoint Class: It is cloudy in Milan

Figure 14.14 Using a service factory

14.5.4 Creating services with multiple implementations

Services can be defined with multiple implementations, which allows a consumer to select an implementation that best suits a specific problem. This is a feature that works best when the service interface provides insight into the capabilities of each implementation class. To provide information about the capabilities of the `IResponseFormatter` implementation classes, add the default property shown in listing 14.36 to the interface.

Listing 14.36 Adding a property in the IResponseFormatter.cs file in the Services folder

```
namespace Platform.Services {
    public interface IResponseFormatter {

        Task Format(HttpContext context, string content);

        public bool RichOutput => false;
    }
}
```

This `RichOutput` property will be `false` for implementation classes that don't override the default value. To ensure there is one implementation that returns `true`, add the property shown in listing 14.37 to the `HtmlResponseFormatter` class.

Listing 14.37 Overriding in the HtmlResponseFormatter.cs file in the Services folder

```
namespace Platform.Services {
    public class HtmlResponseFormatter : IResponseFormatter {

        public async Task Format(HttpContext context, string content) {
            context.Response.ContentType = "text/html";
            await context.Response.WriteAsync($@"
                <!DOCTYPE html>
                <html lang=""en"">
                <head><title>Response</title></head>
                <body>
                    <h2>Formatted Response</h2>
                    <span>{content}</span>
                </body>
                </html>");
        }

        public bool RichOutput => true;
    }
}
```

Listing 14.38 registers multiple implementations for the `IResponseFormatter` service, which is done by making repeated calls to the `Add<lifecycle>` method. The listing also replaces the existing request pipeline with two routes that demonstrate how the service can be used.

Listing 14.38 Defining and using a service in the Program.cs file in the Platform folder

```
//using Platform;
using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

//IConfiguration config = builder.Configuration;

//builder.Services.AddScoped<IResponseFormatter>(serviceProvider => {
//    string? typeName = config["services:IResponseFormatter"];
//    return (IResponseFormatter)ActivatorUtilities
//        .CreateInstance(serviceProvider, typeName == null
//            ? typeof(GuidService) : Type.GetType(typeName, true)!);
//});
//builder.Services.AddScoped<ITimeStamper, DefaultTimeStamper>();

builder.Services.AddScoped<IResponseFormatter, TextResponseFormatter>();
builder.Services.AddScoped<IResponseFormatter, HtmlResponseFormatter>();
builder.Services.AddScoped<IResponseFormatter, GuidService>();

var app = builder.Build();

//app.UseMiddleware<WeatherMiddleware>();

//app.MapEndpoint<WeatherEndpoint>("endpoint/class");

//app.MapGet("endpoint/function",
//    async (HttpContext context, IResponseFormatter formatter) => {
//        await formatter.Format(context,
//            "Endpoint Function: It is sunny in LA");
//});

app.MapGet("single", async context => {
    IResponseFormatter formatter = context.RequestServices
        .GetRequiredService<IResponseFormatter>();
    await formatter.Format(context, "Single service");
});

app.MapGet("/", async context => {
    IResponseFormatter formatter = context.RequestServices
        .GetServices<IResponseFormatter>().First(f => f.RichOutput);
    await formatter.Format(context, "Multiple services");
});

app.Run();
```

The `AddScoped` statements register three services for the `IResponseFormatter` interface, each with a different implementation class. The route for the `/single` URL uses the `IServiceProvider.GetRequiredService<T>` method to request a service, like this:

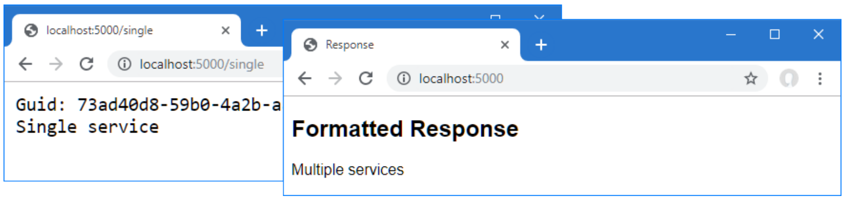
```
...
context.RequestServices.GetRequiredService<IResponseFormatter>();
...
```

This is a service consumer that is unaware that there are multiple implementations available. The service is resolved using the most recently registered implementation, which is the `GuidService` class. Restart ASP.NET Core and request http://localhost:5000/single, and you will see the output on the left side of figure 14.15.

The other endpoint is a service consumer that is aware that multiple implementations may be available and that requests the service using the `IServiceProvider.GetServices<T>` method.

```
...
context.RequestServices.GetServices<IResponseFormatter>()
    .First(f => f.RichOutput);
...
```

This method returns an `IEnumerable<IResponseFormatter>` that enumerates the available implementations. These are filtered using the LINQ `First` method to select an implementation whose `RichOutput` property returns `true`. If you request http://localhost:5000, you will see the output on the right of figure 14.15, showing that the endpoint has selected the service implementation that best suits its needs.



```
Guid: 73ad40d8-59b0-4a2b-a
Single service
```

Formatted Response

Multiple services

Figure 14.15 Using multiple service implementations

14.5.5 Using unbound types in services

Services can be defined with generic type parameters that are bound to specific types when the service is requested, as shown in listing 14.39.

Listing 14.39. Using an unbound type in the Program.cs file in the Platform folder

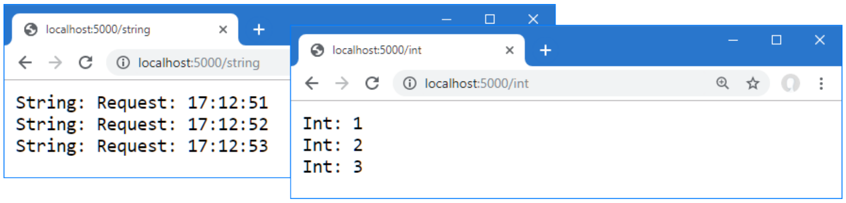
```
//using Platform.Services;

var builder = WebApplication.CreateBuilder(args);

//builder.Services.AddScoped<IResponseFormatter, TextResponseFormatter>();
//builder.Services.AddScoped<IResponseFormatter, HtmlResponseFormatter>();
//builder.Services.AddScoped<IResponseFormatter, GuidService>();

builder.Services.AddSingleton(typeof(ICollection<>), typeof(List<>));

var app = builder.Build();

//app.MapGet("single", async context => {
//    IResponseFormatter formatter = context.RequestServices
//        .GetRequiredService<IResponseFormatter>();
//    await formatter.Format(context, "Single service");
//});

//app.MapGet("/", async context => {
//    IResponseFormatter formatter = context.RequestServices
//        .GetServices<IResponseFormatter>().First(f => f.RichOutput);
//    await formatter.Format(context, "Multiple services");
//});

app.MapGet("string", async context => {
    ICollection<string> collection = context.RequestServices
        .GetRequiredService<ICollection<string>>();
    collection.Add($"Request: {DateTime.Now.ToLongTimeString()}");
    foreach (string str in collection) {
        await context.Response.WriteAsync($"String: {str}\n");
    }
});

app.MapGet("int", async context => {
    ICollection<int> collection
        = context.RequestServices.GetRequiredService<ICollection<int>>();
    collection.Add(collection.Count() + 1);
    foreach (int val in collection) {
        await context.Response.WriteAsync($"Int: {val}\n");
    }
});

app.Run();
```

This feature relies on the versions of the AddSingleton, AddScoped, and AddTransient methods that accept types as conventional arguments and cannot be performed using generic type arguments. The service in listing 14.39 is created with unbound types, like this:

```
...
services.AddSingleton(typeof(ICollection<>), typeof(List<>));
...
```

When a dependency on an ICollection<T> service is resolved, a List<T> object will be created so that a dependency on ICollection<string>, for example, will be resolved using a List<string> object. Rather than require separate services for each type, the unbound service allows mappings for all generic types to be created.

The two endpoints in listing 14.39 request ICollection<string> and ICollection<int> services, each of which will be resolved with a different List<T> object. To target the endpoints, restart ASP.NET Core and request http://localhost:5000/string and http://localhost:5000/int. The service has been defined as a singleton, which means that the same List<string> and List<int> objects will be used to resolve all requests for ICollection<string> and ICollection<int>. Each request adds a new item to the collection, which you can see by reloading the web browser, as shown in figure 14.16.



```
String: Request: 17:12:51
String: Request: 17:12:52
String: Request: 17:12:53
```

```
Int: 1
Int: 2
Int: 3
```

Figure 14.16 Using a singleton service with an unbound type

Summary

- Dependency injection allows application components to declare dependencies on services by defining constructor parameters.
- Services can be defined with a type, an object, or a factory function.
- The scope of a service determines when services are instantiated and how they are shared between components.
- Dependency injection is integrated into the ASP.NET Core request pipeline.